Decision Management Community Challenge Mar 2019
Offering Donated Organs for Transplant (Part 2)

Solution using Stateless Sessions in Drools and corrections to Stateful Solution

(Bob Moore, JETset Business Consulting, 28 Mar 2019)

1 Introduction

To avoid vast amount of repetition, this document had been written as an addendum to my original submission for the challenge at challenge2019mar.bobmoore.pdf, and you need to read that before looking too closely at this. The challenge describes a stateful decision problem, and the original submission describes a solution using stateful sessions in Drools. In this submission I revisit the problem to demonstrate a solution based on stateless sessions in Drools.

Additionally, I did further tests on the original solution where offers for multiple donors were active concurrently. This turned up a few minor problems in the original rules, so these are also described. Ironically, because the stateless solution only involves a single donor at a time running in the rule-engine, the correct operation of the stateless solution does not really depend on these corrections.

2 Moving from Stateful to Stateless

As described in the original submission, if we have a stateful decision problem which reaches an end point (as opposed to being 'always on') there is a fairly simple way to use a stateless approach which involves building a layered decision system having an inner stateless component, wrapped by an outer stateful one which holds and maintains the state. As regards this particular problem the execution logic from a user of the decision system appears to be:

1. Add a new donor into the service
2. Receive back some offers to be made
3. Make the offers
4. When a response to an offer is received send it to the service
5. If the service responds to the response with more offers, make the new offers
6. If there are outstanding offers go back to 4
7. Otherwise the offer process for this donor is complete

Since we only provide the donor information at step 1, the 'state' of the decision must be held within the service. The initial solution achieves this in a transparent manner, as the state is held in Drools' working memory. However, if we run a stateless session, the

working memory is destroyed as soon as we exit the session, so we need to hold it somewhere else. Likewise, when we start a new session (when a new offer response is received) we need to inject it into the session along with the response. So, what is required is for the logic above to be augmented (in a way transparent to the users of the service) giving:

1. Add a new donor into the service
    a. After initialising the donor in the stateless Drools session, persist the donor in the stateful layer
2. Receive back some offers to be made
3. Make the offers
4. When a response to an offer is received send it to the service
    a. Within the stateful layer retrieve the donor and insert it in the stateless Drools session together with the offer response
    b. After the session completes have the stateful layer persist the (updated) donor
5. If the service responds to the response with more offers, make the new offers
6. If there are outstanding offers go back to 4
7. Otherwise the offer process for this donor is complete

Notice that one implication of the change being 'transparent' is we expect to be able to use the same domain model for the stateless solution as for the stateful one, and indeed this is the case.

As was the case in implementing the Stateful solution, some Java wrapper code is needed to create a rule-engine, load the rules and then execute them. By far the 'simplest' place to the persist the donor information is simply in the Java wrapper code. Specifically, as new donor objects appear, we can simply keep a handle on them in the Java, and then later when we get a new response, we have the donor information at hand. Since ideally the solution should support concurrent offers (i.e. offers of organs from multiple donors) we need to be able to match offers to the donors they relate to, something easily done with say a hash table.

While this is simple, it has a downside in that the state remains volatile, so if the program running the decision service goes down, the state on any ongoing offers is lost. As described in the original solution, surviving some level of system failure is one of the major upsides to using a solution based on a stateless rather than a stateful core. To enjoy this advantage though we need to keep the donor in a 'persistent' data store.

The ideal would be a database – we could use something like MongoDB (which is specifically oriented to managing hierarchical data) to store the data in JSON format, but unless we intend to interrogate the structure of the donor outside the service we could just as well push it into a relational DB either as a JSON string or even as a binary object. For the purposes of the solution, a slightly simpler solution to provide persistent storage has been chosen, namely, to save the state to the disk as JSON files.

Now let's think about the rules we need for a stateless compared to a stateful solution. If you look at the rules described in the original submission, there is no 'inference' involved really, so there is no obvious reason why one cannot use the same rules for

both a stateful and a stateless solution[1]. And it turns out that this is the case – well almost!

Looking them over, it quickly becomes obvious we don't need to modify any of the existing rules, but we what we do need is to add an additional one. If you call the rule service to process a new donor, the logic executes fine, but if you call the rule service again to process a response (passing in both the response and the current state of the donor) it will fail. Most of the objects needed are not available to the rules because the first initialise rule (which inserts the donor and its subcomponents into working memory) does not fire (since the donor status is 'in progress' not 'new').

Setting the donor status back to 'new' before invoking the rules doesn't help as this then will result in additional initialise calls and additional (unwanted) updates to the state. The fix is easy though. One just adds in an extra rule to insert the donor and its subcomponents into working memory for an existing state. For this to fire we set the status to a new value 'from DB' before inserting it into the working memory. This new rule is basically a copy of the first initialise rule with some minor modifications as noted below.

It's probably also worth pointing out that the 'Tidy up' rules in the stateful solution are redundant for the stateless solution, all working memory is discarded following execution. As the solution stands they still execute, but ideally they would not even be loaded when using stateless sessions.

3 The Rules

3.1 The Stateless Reinitialise Rule

The only new rule needed to support stateless execution of the rules looks as follows:

```
rule "Initialise from DB - Add Objects to Working Memory"
  salience 100
  when
    $donor: Donor(status == Constants.FROM_DB)
  then
    insert($donor.getNextAction());
    for(Organ organ: $donor.getOrgans().values()) {
      insert(organ);
    }
    for(CandidateList candidateList: $donor.getCandidateLists().values()) {
      insert(candidateList);
      for(Candidate candidate: candidateList.getCandidates()) {
        insert(candidate);
      }
    }
    modify($donor) { setStatus(Constants.IN_PROGRESS) }
end
```

As described above it is very similar to the rule "`Initialise - Add Objects to Working Memory`" of the 'Initialise Rules' (but as below the original version of this has been updated slightly). The main difference is the donor status tested in the conditions, and donor status set in the actions, which ensures we do not execute any of the 'Initialise Rules'.

[1] Drools explicitly links the ideas of stateful and inference and limits some kinds of logical reasoning in stateless sessions. See discussion for further comments.

Note that when inserting a donor, we move from the 'Initialise Rules' directly to the 'Next Action Rules'. When we are processing a response, we move from the 'Stateless Reinitialise Rule' to the 'Response Rules' before reaching the 'Next Action Rules'.

3.2 Corrections to Original Rules

The problem with the original rules seemed to stem from the way Drools manages the '**forall**' test where no objects satisfy the filter conditions. Thinking back, I think I may have experienced this before when using Drools on a previous challenge.

Predicate logic relates the existential ($\exists$) and universal ($\forall$) quantifiers by treating the statements $\forall x \in A : P(x)$ and $\neg\exists\, x \in A : \neg P(x)$ as equivalent (or if you prefer it in English, the statement *'all elements in set A satisfy condition P*' is equivalent to the statement '*there are no elements in the set A which do not satisfy condition P*'). Drools seems to respect this equivalence if the set of objects considered has an element in it but gives different answers if it does not.

If there are no objects in the 'interesting' set and we use the '**forall'** form (corresponding to $\forall x \in A : P(x)$), the condition is always false, while if we use '**not exists**' (corresponding to $\neg\exists\, x \in A : \neg P(x)$ the condition is always true – which is what predicate logic tells us is it should be (the argument being that 'not exists' means that there are no elements in the set which satisfies the condition. And if there are no elements in the set to start with, the condition is clearly satisfied).

3.2.1 The Initialise Rules

When trying to work out what the problems with **forall**, it occurred to me that the logic of the two rules "`Initialise - Offer Organ with HP List`" & "`Initialise - Offer Organ if it is the only one`" could be much more simply handled, by setting a default state of 'offering' while inserting the objects into working memory. So, I deleted these two rules and updated the rule "`Initialise - Add Objects to Working Memory`" to look like:

```
rule "Initialise - Add Objects to Working Memory"
  salience 100
  when
    $donor: Donor(status == Constants.NEW)
  then
    insert($donor.getNextAction());
    for(Organ organ: $donor.getOrgans().values()) {
      organ.setStatus(Constants.OFFERED);                    ← New action added
      insert(organ);
    }
    for(CandidateList candidateList: $donor.getCandidateLists().values()) {
      insert(candidateList);
      for(Candidate candidate: candidateList.getCandidates()) {
        insert(candidate);
      }
    }
    modify($donor) { setStatus(Constants.INITIATING) }
end
```

Another minor change was to reduce the salience of the "`Initialise - Generate dummy offer response`" rule from 100 to 99. This was not critical in the sense that the default rule execution semantics should ensure this rule fires after the rules above it, and before the one which follows but it does make clear that it has to fire after the other

rules to ensure the correct state of the organs has been set before generate dummy **offeResponse** instances for the ones which will be 'offered'.

Finally, the rule "`Initialise - Mark donor offering in progress`" was causing problems because of the **forall** in the last condition. Initially I changed this to the **not exists** formulation, and everything worked. But when I updated the rule "`Initialise - Add Objects to Working Memory`" the condition became redundant, so I simply removed it altogether. Having updated the salience of the preceding rule, it was also prudent to reduce the salience of this rule to 98, so it now looks like:

```
rule "Initialise - Mark donor offering in progress"
  salience 98                                    ← salience updated
  when
    $donor: Donor($donorId: donorId, $nextAction: nextAction, status == Constants.INITIATING)
                                                 ← second condition removed
  then
    modify($donor) { setStatus(Constants.IN_PROGRESS) }
end
```

3.2.2 The Next Action Rules

With the changes made to the initialise rule, it turned out the only place that **forall** was now used was in the rule "`Next Action - Complete`". The **forall** was replaced by the equivalent **not exists** form to give the modified version:

```
rule "Next Action - Complete"
  when
    $donor: Donor( $donorId: donorId, $nextAction: nextAction,
      status == Constants.IN_PROGRESS)
    not(exists(Organ($donorId == donorId,                ← forall replaced with not exists
      status != Constants.ACCEPTED && status != Constants.UNAVAILABLE)))
  then
    modify($donor) { setStatus(Constants.COMPLETE) }
    modify($nextAction) {setStatus(Constants.COMPLETE) }
end
```

And all the problems were solved!

4 From Rules to a Service – Framework for Stateless Solution

Updating the rules to support stateless sessions proved very simple. Doing the same for the framework involves more effort, because one has to create some logic, which a Drools stateful session does for us automatically. Fortunately, not a huge amount of extra work is involved.

Drools stateless service are activated in a different manner to stateful ones. In a stateful service one inserts some objects and calls 'fire all rules' on the stateful session with some Java code which looks like this:

```
kStatefulSession.insert(object);
kStatefulSession.fireAllRules();
```

One can call '`insert`' as many times as one likes with the stateful session, before triggering the rules. However, in a stateless service, one inserts the objects and then trigger the rules in a single '`execute`' call:

```
kStatelessSession.execute(objects);
```

This is a bit irritating, since for the stateless solution one only ever needs to insert one object (a donor on the first call, a response on subsequent ones), while for the stateless session, one inserts one object on the first call, but two on the second (the donor – providing the current state, and the response). Fortunately, the 'execute' method will take a collection of objects, so instead of calling 'insert' one creates a collection and adds all the required objects before calling 'execute'.

The code for the initial processing of a donor is pretty simple once we have created the initial donor we just need to pass it to a to call 'execute'. and save the state after it returns as follows:

```
kStatelessSession.execute(donor);
Utils.saveObject(donor.getDonorId(), donor);
```

When processing a response to an offer first we retrieve the persisted state from the disk. Then we need to update the donor status to 'from DB' so the rule-engine will recognise the donor as already having had its initial processing completed and fire the new rule "Initialise from DB - Add Objects to Working Memory". Next, we build a collection (of only two objects), at which point we can pass the collection to the rule-engine with a call to 'execute'. Finally, when the rule engine is finished, we persist the updated state back to the disk. The following code fragment shows how all this is done and it is the stateless equivalent of the calls to 'insert' and 'fireAllRules' in Java for the stateful solution shown above.

```
donor = Utils.loadDonor(donor.getDonorId());
donor.setStatus(FROM_DB);
ArrayList<Object> objectsToInsert = new ArrayList<Object>();
objectsToInsert.add(donor);
objectsToInsert.add(offerResponse);
kStatelessSession.execute(objectsToInsert);
Utils.saveObject(donor.getDonorId(), donor);
```

The code is a bit more long-winded but given we have moved the state management out of the rule-engine into the Java framework a total of an extra six lines of code is surprisingly little[2].

Having run though a substantial number of the 50+ variations of the basic 8 test scenarios, the stateless solution consistently provides the same (and the correct!) outcomes as running with the corrected stateful version.

5 Discussion

As the solution demonstrates, moving from an in memory stateful decision service to a persistent store stateful service wrapping a stateless decision service can be a quite

[2] One useful time-saver was that I had already written load/save logic for test purposes when creating the stateful solution – so I could reuse the utility methods *loadDonor* & *saveObject* for persisting state.

painless process provided the stateful service has a definitive end point (typically these kinds of problems are characterised as having ‘dialog’ like interactions).

The ease of doing this in Drools may to some degree rely on the rules for this problem not involving ‘inference’ as such. Drools explicitly links the ideas of stateful and inference and limits some kinds of logical reasoning in stateless sessions. For other tools (e.g. Blaze Advisor), ‘stateless’ merely signifies that all state is lost when the session terminates but this puts no limit the internal rules capabilities. I have argued in a previous posting on the DMC site[3], a significant number of important ‘business decisions’ (whatever they may be) are ‘inference free’ so this may not a be real issue in most simple stateful problems of this nature.

To be honest, I regard the fact that the virtually the same set of rules work for both modes as a bit serendipitous, I had expected I would need to do rather more work. It’s also worth perhaps observing normally one would make a specific choice on whether to use a stateless or stateful approach before starting on writing the rule code. An interesting question, but one I cannot in all honesty answer is if I had started by building a solution around stateless session to start with would I have ended up with the same set of rules? Yes, they would probably have been very similar – but would there have been any material differences? Would have migrating them to a stateful session have been so painless?

6 Source Code

As per the original submission I’ve not included the source of the Java code and Rules files for this solution but if you want to see all the gory details drop me a line and I’ll be happy to e-mail over a copy.

[3] See https://dmcommunity.org/2018/04/09/does-real-world-decision-management-need-inference/#more-4144